A Case for Nano-Kernels

See-Mong Tan David K. Raila Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield
Urbana, IL 61801
{*stan,raila,roy*}@cs.uiuc.edu

Abstract

The μ*Choices* operating system splits the microkernel into a machine-independent part and a machine-dependent sub-microkernel. The sub-microkernel, called the *nano-kernel* in μ*Choices*, encapsulates the hardware and presents an idealized machine architecture to the rest of the system. Higher levels of the system access the nano-kernel through a single interface. Nano-kernels are useful because they significantly enhance portability of the entire microkernel. The interface provided by the idealized machine architecture of the μ*Choices* nano-kernel is a good match for constructing higher-level abstractions in the machine-independent microkernel. In the μ*Choices* nano-kernel, we have *fully decoupled* the nano-kernel from all higher-level abstractions. Thus, it should be possible to construct many *different* operating systems on top of the nano-kernel. The μ*Choices* nano-kernel is built as an object-oriented framework. The framework guides porting to new hardware platforms, and allows the specialization of its components for efficient, machine-specific implementation.

1 Introduction

A prime concern in modern operating systems is system portability across different hardware platforms. Operating system implementors have migrated from coding in pure assembler to writing most parts of an operating system in high level languages such as C[11, 9]. This enhances portability by reducing the amount of machine-dependent assembler that needs to be rewritten for every new port of the operating system.

Coding in high level languages has indeed made modern operating systems code much clearer, more flexible, and easier to maintain. However, significant amounts of machine-dependent details remain in even the high level portions of the operating system. Such machine dependencies may appear in many major subsystems of an operating system, such as the memory management or process subsystems. The virtual memory subsystem requires machine-dependent code for handling the hardware memory management unit (MMU). The process subsystem requires both the MMU code and code for handling processor context saves and restores. Interrupt handling is another case in point.

In the μ*Choices* operating system[2, 10], we have split the kernel into two portions. The machine-dependent *nano-kernel* encapsulates the physical hardware and provides hardware support for the rest of the machine-independent microkernel. It provides the microkernel with the needed mechanisms for implementing higher-level abstractions, such as processes, timers, and virtual memory. The nano-kernel is not a wrapper around assembler routines. Because μ*Choices* is an object-oriented operating system, the nano-kernel is built as a framework of classes that captures the essential properties of the low-level hardware, presenting a useful interface to the higher levels of the kernel in a machine-independent way. It provides the building blocks for constructing kernel abstractions.

Our thesis is that sub-microkernels for hardware support, such as the μ*Choices* nano-kernel, greatly benefit operating system construction. We enumerate the reasons below. They are useful because they significantly enhance portability. The object-oriented framework upon which the μ*Choices* nano-kernel is based guides and simplifies the process of porting the operating system to new hardware platforms. We describe the port of the nano-kernel to two highly dissimilar platforms: the first an emulation of a symmetric shared memory multiprocessor on top of Unix, and the second to the Sun SPARCstation.

The decomposition of the μ*Choices* kernel into a machine-independent part and a sub-microkernel has allowed us to factor out all hardware dependencies in the various subsystems. This leads to easily portable higher levels. We also show that the interface provided by the idealized machine architecture of our nano-kernel is a good match for constructing the higher-level abstractions in the machine-independent microkernel. Because a good match is possible, a careful design of the sub-microkernel can negate any harsh performance penalties. In addition, we have *fully decoupled* the μ*Choices* nano-kernel from all higher-level abstractions. Thus, it should be possible to construct many *different* operating systems on top of the sub-microkernel. Since most operating systems deal with the same abstractions at the machine-independent level (processes, VM, etc.), a single idealized interface can suit many different operating systems. If a standard were agreed upon, the operating system community could exchange sub-microkernels and vastly extend the range of our supported hosts. Finally, decomposing an OS kernel in this manner led us to fully separate the truly machine-dependent components from the machine-independent abstractions used by operating systems.

In the next section, we discuss the μ*Choices* nano-kernel in detail. Then in section 3, interface issues are considered, and we argue that the nano-kernel interface can indeed suit the needs of most general purpose operating systems. The nano-kernel has been ported to an emulation on top of Solaris and natively to the Sun Sparc, and this is described in section 4 We present performance measurements of nano-kernel for two ports in section 5. Related work and the conclusion is given in section 6 and section 7.

2 Nano-Kernel

μ*Choices* is a redesign of the original *Choices* object-oriented operating system[1]. The operating system is composed of independent modules. Modules are implemented as independent object-oriented frameworks that interact through well defined interfaces. Within a framework, the subclassing of components provides the ability to customize its various parts to support different implementations and optimizations.[8]. Modules interact only through well-defined interfaces. Code reuse through inheritance is used within modules internally, and interface reuse through inheritance of abstractions is used between sub-frameworks.

The μ*Choices* nano-kernel is the hardware support module. The nano-kernel encapsulates the data representations and algorithms associated with the instruction set, and virtual address mechanisms of the computer. The μ*Choices* nano-kernel is composed of:

- the *boot* component, responsible for booting and initializing the operating system.
- the *processor* component, responsible for managing physical CPUs.
- the *memory management* component, responsible for managing the MMU, TLB, and virtual address mappings.
- the *exception* component, responsible for handling hardware traps and interrupts, scheduling the software handlers for these events, and for providing a machine-independent interface to these services.
- the *boot console* component, responsible for console output at boot time.
- the *debugger* component, responsible for debugger hooks in the kernel.
- the *interface* component, responsible for providing a single interface for accessing the hardware support module

The code structure for the nano-kernel organizes the above components into four groups:

1. Framework
2. Processor Dependent
3. Machine Dependent
4. Interface

2.1 Framework

The Framework group holds the basic, abstract classes in the nano-kernel for the μ*Choices* idealized machine architecture. These abstract classes reify hardware entities, such as the processor (class `CPU`) and memory management unit (class `MMU`), as well as events, such as interrupts (class `Exception`) and services, such as locks (class `Lock`).

2.1.1 Exceptions

The abstract class `Exception` reifies the interrupts that are generated by the hardware. `Exception` is subclassed in the Processor Dependent and Machine Dependent parts of the nano-kernel for exceptions particular to the target machine's processor and machine type. Table 1 shows the abbreviated methods

Methods		Description
Instance	`stack`	Stack an exception on another
	`raise`	Subclass-defined: called when interrupt occurs
Class	`setExceptionHandler`	Set handler for machine-independent type

Table 1: Methods for abstract class `Exception`.

exported by the `Exception` class. The subclass-specific constructor registers a new `Exception` object in a subclass-specific way, binding it to the hardware such that the `raise` method is called when an exception of that nature occurs. `Raise` is redefined by concrete subclasses. Exceptions may be stacked through the `stack` method. In this case the `raise` method will be called in turn for each stacked `Exception` object when the interrupt occurs. Six *machine-independent exception types* can be registered with the Exception class through the `setExceptionHandler` class method. The machine-independent exception types correspond to the exceptions widely caught by operating system kernels. These are:

1. `MemoryException`, for memory access violations,
2. `IllegalInstructionException`, for invalid instructions,
3. `FloatingPointException`, for arithmetic errors,
4. `TimerException`, for timer expirations,
5. `FreeRunningTimerException`, for free running timer expirations,
6. `IOException`, for IO device exceptions.

Each machine-independent exception type may have a *hardware-dependent vector* associated with it. In most cases, the hardware vector is known in the processor dependent or machine dependent subclass for that exception type. For example, the SPARC processor uses vector 26 for timer expirations. The vector argument is usually required for the type `IOException`, as different IO devices may raise interrupts on different vectors. When `raise` is called on an instance of a concrete subclass of `Exception`, it converts the hardware interrupt to which it is bound into a call to the user installed exception handler for its machine-independent type.

2.1.2 Locks and CPUs

Class `Lock` in the nano-kernel framework implements a mutual exclusion lock between CPUs. Locks form the basis on which higher level abstractions such as semaphores may be constructed as part of a process management subsystem. Locks function in cooperation with the `CPU` class (table 2), which implements an abstract protocol for handling interrupts and other CPU related mechanisms.

An opaque handle of type `InterruptStorage` is exported by the concrete subclass of `CPU`. It represents machine-dependent interrupt mask information.

Method	Description
`startProcessors`	Start all processors running
`disableInterrupts`	Disable interrupts on this CPU
`restoreInterrupts`	Restore interrupts on this CPU
`addToInterruptMask`	Fill mask for machine-independent type
`spinMutex`	Spin lock
`releaseMutex`	Release spin lock

Table 2: Methods for abstract class `CPU`.

`CPU` exports the method:

```
void addToInterruptMask(InterruptStorage& mask,
                        ExceptionType type,
                        int vector);
```

By calling the above method, a client of CPU may construct an interrupt storage mask by specifying the machine-independent exception types and associated hardware-dependent vectors (if necessary). Interrupt storage masks are acted on only by the concrete subclass of CPU, thus on hardware that only supports the masking of interrupt levels, InterruptStorage is an integer value, and addToInterruptMask sets its value to the maximum of its current value and the value corresponding to the associated machine-independent exception type.

Methods	Description
Lock(InterruptStorage& mask)	Create a lock with an interrupt mask
acquire	Acquire a lock
release	Release a lock

Table 3: Methods for class Lock.

A mask is passed as an argument to the constructor of Lock. The mask is used to specify what interrupts to mask out when the lock is acquired. On calling acquire, the lock first masks off the interrupts by calling disableInterrupts on the CPU class. It then uses spinMutex on the CPU class to acquire a spin lock. The reverse procedure of releaseMutex followed by restoreInterrupts is used to relinquish a spin lock. Methods spinMutex and releaseMutex are implemented by the concrete subclasses of CPU.

2.1.3 Memory Management

Tables 4 and 5 illustrate the methods used by the memory management framework for classes Address-Translation and MMU respectively.

Methods	Description
addMapping	Add a mapping from a VM address into a chain of pages
removeMapping	Remove VM mapping
changeProtection	Change VM protection for VM range
changeStatus	Change status of mappings for VM range
reference	Get reference information

Table 4: Methods for class AddressTranslation.

Methods	Description
enable	Enable the MMU
activate	Activate an AddressTranslation
flushCache	Flush the MMU cache for an address range

Table 5: Methods for class MMU.

VM mappings for a VM address space is constructed by adding chains of physical pages into an instance of AddressTranslation. The collection of pages in an instance of the class represents a virtual memory domain.

AddressTranslation is an abstract class. Methods on the class allow one to add or remove mappings to physical pages at different addresses, as well as affect the protection levels of the pages in the translation. Although the protocol is specified by the abstract class, the actual implementation is left up to a concrete subclass. Concrete subclasses implement the actual translation in an efficient, machine-dependent manner.

Class MMU implements an abstract protocol for controlling hardware memory management units. Instances of class MMU encapsulate the memory management units on the machine. MMUs operate on instances of the AddressTranslation class. The basic methods exported by the MMU interface allow one to enable its operation, activate a given address translation and flush the MMU cache for an address range. A virtual memory domain is mapped when the activate method of an instance of class MMU is invoked with the translation.

2.1.4 Processor Contexts

The `ProcessorContext` class gives the higher level microkernel the means to implement a process subsystem, with methods to checkpoint and restore the running CPU thread. Processor contexts store the state of the CPU, including the program counter, registers, interrupt mask information, stack pointers, frame pointers and any other information that the CPU may need to restore a running task. The Processor Dependent group implements the concrete subclass of `ProcessorContext` for specific processor architectures. VM mappings are not included in processor contexts. Higher level process management subsystems mamnage that information with instances of the `AddressTranslation` class (table 4).

Methods	Description
`checkpoint`	Save present CPU context
`restore`	Restore previously saved context

Table 6: Methods for class `ProcessorContext`.

Two methods are exported, namely `checkpoint` and `restore`. The first saves the present context, the other restores a previously saved context. The `checkpoint` returns 0 on the first call, and will appear to return non-zero on being restored by another thread.

2.1.5 Console

The `Console` implements a C++-like output stream. The machine-dependent group implements the actual mechanics of writing characters to the output console. Output is required for debugging, thus we have included a console at the lowest level of the nano-kernel.

2.1.6 System Configuration

Methods	Description
`numberOfProcessors`	Return the number of physical processors
`pageSize`	Return the machine page size
`physicalMemoryMap`	Return a map of physical memory
`bind`	Bind a key to a value
`unbind`	Unbind a key from a value
`lookup`	Lookup a value with a key

Table 7: Methods for class `SystemConfiguration`.

An instance of the `SystemConfiguration` abstract class allows the querying of system parameters such as the number of processors, the machine page size, and amount of RAM. This is illustrated in table 7. The `bind`, `unbind` and `lookup` methods permits the addition of miscellaneous configuration information, which may be machine-dependent.

2.2 Processor Dependent and Machine Dependent

The Processor Dependent group contains implementations of abstract classes in the framework for particular processor architectures. For example, within ProcessorDependent, the subclasses for the SPARC or MIPS processors may be found. The framework requires concrete subclasses for the abstract classes

- CPU,
- MMU,
- Exceptions,
- AddressTranslation, and
- ProcessorContext.

The Machine Dependent group contains subclasses for particular machines and specializations between machines of the same processor architecture. The group holds concrete subclasses of the abstract classes for

- Console,
- MemoryAllocator,
- Exceptions,
- SystemConfiguration.

Specialized boot code for the machine is found here as well. Booting occurs in a machine-dependent manner in the nano-kernel, and proceeds through nano-kernel initialization as specified in the abstract class framework. The micro-level is entered through the invocation of a well-known entry point (`KernelMain()`) at the end of the nano-kernel's initialization. Further interaction between the micro-level and the nano-kernel occurs through the nano-kernel interface (discussed in section 3).

Porting from one machine to another is thus concentrated in realizing the concrete subclasses within the Processor and Machine Dependent groups. The interactions between components in the nano-kernel framework is already specified by the abstract classes in the Framework group. Our experience indicates that the process of porting is simplified when one works within a well engineered framework of classes.

3 Interface Design

In the design of the nano-kernel interface, one must take care to craft it at the appropriate level. Too abstract or "high" an interface restricts the flexibility of the machine-independent part of the kernel. Too low an interface does not buy one very much, and may let machine-dependencies creep into the higher levels. Related to this is what machine-dependencies may be factored out of the higher levels of the kernel. We have taken care to design the nano-kernel such that useful, *machine-independent* mechanisms are provided for constructing micro-level abstractions such as

- processor allocation and scheduling,
- process management and synchronization,
- physical and virtual memory management, and
- interrupt processing.

The Interface group holds the definition of the interface that the nano-kernel exports. We allow access to the nano-kernel only through this interface. An abbreviated list of interface methods and descriptions is shown in table 8.

One notices that the methods in the interface correspond closely to the methods exported by individual classes in the nano-kernel framework. Rather than make the individual nano-kernel objects directly available to the micro-level, we interpose an explicit interface between the higher levels and the nano-kernel for the following reasons. First, the interface enforces module independence between the micro- and nano-levels. No components are exported beyond the nano-kernel, only opaque handles. This architecture decouples the dependence of the micro-level on the specific nature of the classes and objects in the nano-kernel, permitting a high degree of evolutionary flexibility as the nano-kernel itself is changed and extended in its lifetime. For example, locks in the nano-kernel can be changed without affecting the rest of the system. Second, an explicit interface also permits us to capture and limit the way the nano-kernel is accessed. The tight coupling of components within the nano-kernel requires that classes publish methods that are used for communication between components in the nano-kernel, but are not essential for the micro-level to know about. The interface prevents the micro-level from manipulating nano-kernel objects in undesirable ways. Third, the interface may itself be extended to allow the aggregation in one interface call to multiple nano-kernel calls. This particular object-oriented design pattern is called the Facade[6].

The cost of an explicit interface is the overhead of one function call per invocation of the nano-kernel.

3.1 Suitability of the Interface

We suggest that the above interface is suitable for a wide range of general purpose operating systems. Operating systems may differ, but most deal with the same abstractions:

- **processes, process management and synchronization**

  Time-shared processor allocation is governed by all modern operating systems. In order to implement processor allocation, kernels require ways to catch timer interrupts, checkpoint the current running context and switch to another context. Virtual memory mappings may or may not be

Method	Comment
`cpuId`	Get id of current CPU
`startProcessors`	Start all CPUs
`cpuDisableInterrupts`	Interrupts off this CPU
`cpuRestoreInterrupts`	Restore interrupt mask
`addToInterruptMask`	Add machine-independent exception type to interrupt mask
`timerOn`	Set timer value
`newLock`	Get mutual exclusion lock
`deleteLock`	Delete mutex lock
`lockAcquire`	Acquire the lock
`lockRelease`	Release the lock
`newContext`	Create a new processor context
`contextCheckpoint`	Checkpoint to context
`contextRestore`	Restore saved context
`newTranslation`	Create a new address translation
`addMapping`	Add a mapping to a translation
`enable`	Enable a MMU for a CPU
`addTranslation`	Add a translation to the MMU
`changeProtection`	Change VM protection
`changeStatus`	Change status bits
`flushCache`	Flush MMU cache
`reference`	Get reference bits for page
`setExceptionHandler`	Set a handler for a machine-independent exception type
`physicalMemoryMap`	Get the map of physical memory
`numberOfProcessors`	Return number of CPUs
`pageSize`	Return size of a page
`lookup`	Lookup a value with a key

Table 8: Interface to the Nano-Kernel.

changed on a context switch. The nano-kernel supports a process subsystem by giving the higher level the processor context, timer exception, and address translation components to work with. Figure 1 depicts the relationships between two processes in the same virtual memory space, together with their associated processor context and address translation objects in the nano-kernel for the *μChoices* operating system. Processes are represented as instances of class `Process` in the process subsystem, and a virtual memory space is represented by an instance of class `Domain` in the VM subsystem.

- **physical and virtual memory management**

  An operating system kernel must keep track of the allocation of physical pages and the virtual memory mappings for various VM spaces. The nano-kernel does not impose any predefined memory management strategy, but provides access to a physical memory map in the interface. In *μChoices* the map is used in the microkernel to construct the `Store`, an allocator of physical memory pages. VM mappings may be constructed through address translations in a machine-independent way. The interface hides the details of address translations. The actual translations are stored by concrete subclasses in efficient, machine-dependent implementations.

- **interrupt processing**

  While a multiplicity of interrupts may be generated and differ from one machine to another, most operating systems are only interested in a very few at the micro-level. These were enumerated in section 2.1.1. Thus we can provide a list of machine-independent types and construct the microkernel based on these. The mapping of machine-independent type to hardware vector is accomplished by the concrete, processor or machine-dependent subclasses of `Exception` in the nano-kernel.

  IO interrupts caused by IO devices cannot easily be cast into a machine-independent mold. In this case, the nano-kernel provides an optional `vector` argument that its device driver clients may use in order to register handlers. Since device drivers are inherently machine-dependent anyway, this does not compromise the machine-independent nature of the rest of the interrupt processing interface.

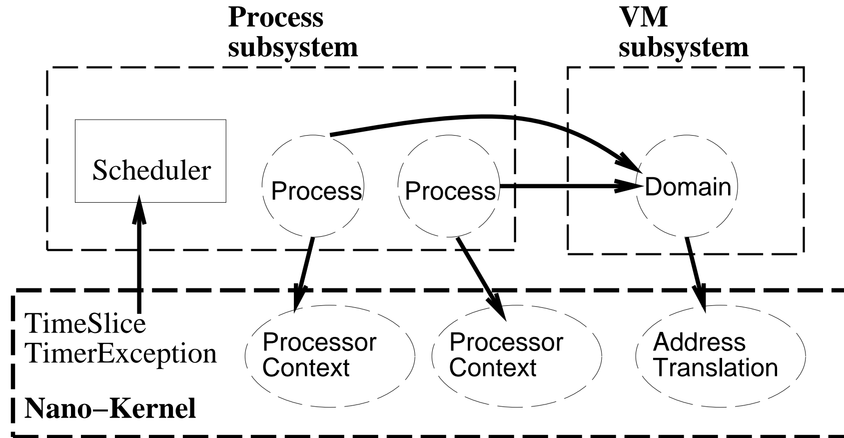

Figure 1: Nano-Kernel, Process and VM Subsystems, with two processes sharing the same VM space.

While the interface in its present form is sufficient for the current version of *µChoices*, other operating systems may rely on extra machine-dependent features, for example, aggressive cache-line placement of dynamically allocated objects. In this case the primary interface can be extended with a method that invokes a nano-kernel allocator. The allocator would be specialized in the Machine Dependent group for efficient placement on a particular target architecture.

4 Two Ports of the Nano-Kernel

The nano-kernel has been ported to two dissimilar platforms. The first provides an emulation of a symmetric shared memory multiprocessor on top of the Solaris flavor of Unix. The second is a native port to the Sun SPARCstation.

µVirtualChoices is an implementation of the *µChoices* nano-kernel that emulates a symmetric shared memory multiprocessor on top of Solaris. *µVirtualChoices* provides a convenient and robust prototyping environment for testing and debugging design ideas in *µChoices* similar to *VirtualChoices*[3]. It has a much faster edit-compile-test cycle, no need for dedicated hardware, and is compatible with native implementations of *µChoices*. This section briefly presents the concrete subclasses that *µVirtualChoices* supplies to implement *µChoices*. The *CPU* subclass in *µVirtualChoices* is programmed with a UNIX process. Additional CPUs are programmed as additional UNIX processes communicating via shared memory and the underlying Solaris filesystem. On a multi-CPU machine true parallel execution can be achieved. Locking is implemented with test-and-set or swap instructions in the SPARC instruction set. The *MMU* is realized through the virtual memory environment of the Solaris process. The memory mapped file facilities are programmed to emulate the manipulation of the `AddressTranslation` and a Solaris file provides the representation of the pages of physical memory. Interrupts and traps in *µVirtualChoices* are realized through the use of Unix *signals*. Signals bear a close resemblance to hardware interrupts on physical processors and can be programmed to emulate interrupts. The signals are programmed by registering handlers, catching signals associated with IO, virtual memory, timers, instruction faults, etc., and by manipulating the Unix signal mask to mask and unmask interrupts.

The SPARC native port of the nano-kernel subclasses the same abstract classes in the nano-kernel Framework for the SPARCstation. The higher levels of our microkernel, namely the Kernel, Process, Virtual Memory, Timer, and Network subsystems, are the same between the two ports.

5 Performance

We measured the performance of several key functions in the nano-kernel. We also measured context switching performance of the Process subsystem in *µChoices*. Process context switching relies on the facilities provided by the nano-kernel, including the use of processor contexts and locks. The performance is summarized in table 9 for *µVirtualChoices* on the Sparcstation 600MP, and the Sparcstation 2 in native mode. The percentages in parantheses indicate the overhead over similar operations in the original

Operation	**Time (μs)**	
	VChoices	**Sparcstation 2**
Lock acquire and release	59 (0.3%)	5.7 (7.5%)
Exception conversion	0.2	0.4
Context switch (kernel threads)	348.4 (0.1%)	76.8 (1%)

Table 9: Nano-kernel performance measurements. Percentages indicate overhead compared to original *Choices*.

Choices operating system. Lock acquistion and release in *μVirtualChoices* requires system calls to set the process signal mask, but only requires interrupt masking for the native Sparcstation. Exception conversion from hardware-raised interrupts to a machine-independent exception type is similar in both ports. No comparable mechanism is available for comparison with in the orignal *Choices*. Context switching is a higher-level function implemented by the Process subsystem and involves disabling interrupts, adding the current process to one of several possible process queues, retrieving the next ready process from the run queue, computing time slice residuals, switching the processor context, and restoring the previous interrupt mask.

The overhead is low for calling interface functions directly, and negligible for context switching, a higher level function composed out of nano-kernel primitives.

6 Related Work

There have been many attempts at defining minimal kernel services in modern operating systems. MIT's Exokernel[4] is intended as a minimal kernel that directly exposes the hardware capabilities of the machine, leaving traditional OS abstractions up to implementations in user-level libraries. Lipto's[5] *nugget* is a "truly minimal kernel", managing low-level resource allocation mechanisms such as processor allocation, memory management and interrupt processing. Spring's[7] *nucleus* is similar, supporting domains (virtual address spaces), threads and doors (which handle object-oriented calls between domains).

The nano-kernel is at a lower level than all of these approaches. For example, in the Exokernel, user programs or their libraries may request time slices from the Exokernel, thus processor allocation is directly controlled by the Exokernel itself and is not available for manipulation at higher levels. In contrast to providing processor allocation, the nano-kernel provides the *mechanisms* for constructing processor allocation strategies in a machine-independent way. Higher level process management subsystems may register handlers to catch timer exceptions. Process objects may be constructed from combining references to nano-kernel provided processor context and address translation objects. In contrast to providing memory management, the nano-kernel provides the mechanisms for manipulating virtual address translations together with methods for enabling and disabling maps on the hardware MMU. In the nano-kernel interrupt processing is split between a machine-dependent part handled by the `Exception` class together with its concrete processor and machine-dependent subclasses, and a machine-independent part which the higher levels refer to via machine-independent exception types.

The port of the nano-kernel to Solaris is similar to Nachos[12]. Nachos is a multithreaded operating system simulated as a regular Unix process. It was developed for pedagogical purposes. *μVirtualChoices* is different in that we use *real* OS code that is not stripped or simplified. *μVirtualChoices* also provides multiprocessor support, with physical processors simulated as separate Unix processes. This allows simulated CPU separate memory management capabilities. Application code is also not interpreted, and page faults are caught through signal handlers for memory segmentation and bus violations. This provides a more realistic simulation of hardware. *μVirtualChoices* supplies a complete prototyping environment for our operating systems development.

7 Conclusion

The nano-kernel in *μChoices* is the hardware support module that localizes machine dependencies and presents an idealized machine architecture to the rest of the operating system. It is entirely divorced from the higher levels of the kernel. Efficient primitives are exported to the rest of the kernel so that higher level operating system abstractions may be constructed. The nano-kernel mechanisms support the construction of micro-level processor allocation, process management, synchronization, virtual memory

management, and interrupt processing strategies. Portability is enhanced since porting to a new machine means working with a single, small, modular system. The framework defined by the abstract classes and relationships in the nano-kernel guide and simplify the process of targeting new hardware. Higher levels of a kernel are portable since there are no residual machine-dependencies.

We posit that many different operating systems may be constructed on top of the portable nano-kernel. This is possible since the abstractions that most operating systems deal with are similar: process, scheduling, virtual memory and interrupt handling. While the basic interface may remain the same, it is likely that it will be extended to accomodate more machine-dependent features that operating systems attempt to utilize.

Constructing the nano-kernel as an object-oriented framework has several advantages. First, the abstract protocols and interactions between components in the nano-kernel framework are well specified and are inherited by the different ports of the system. This makes porting the OS to another machine considerably simpler. Second, the abstract classes in the framework are specialized through object-oriented inheritance to obtain efficient implementations for specific machines. Classes, such as `AddressTranslation`, are subclassed in the nano-kernel into machine-specific ones, such as `SparcTranslation` for the SPARC processor. Thus we avoid the penalty of having abstractions at too "high" a level. `SparcTranslations` hold virtual address translations in a SPARC-specific way and no conversions are required from a machine-independent to a machine-specific representation. While we do convert between machine-dependent and machine-independent exception types, the cost of a conversion is on the order of one function call.

We have attempted to verify the flexibility of our nano-kernel framework and the suitability of its interface with ports to two highly dissimilar platforms. The first is a Unix emulation of a symmetric shared memory multiprocessor, and the second is a native port to the Sun SPARCstation. The code in the micro-levels of *μChoices* is shared, unchanged, between both ports. The measured overhead of an explicit nano-kernel layer, compared to our previous *Choices* implementations on the same platforms, is marginal.

8 Acknowledgements

A description of the *μChoices* nano-kernel appeared as a short position paper[10]. We thank the excellent comments by our anonymous reviewers on that paper. Amitabh Dave, Willy Liao, David Putzolu, Tin Qian, Aamod Sane, Mohlalefi Sefika, Ellard Roush, and Lun Xiao, members of the Systems Research Group at the University of Illinois, also contributed extensively to the design of *μChoices* and the nano-kernel.

References

[1] Roy H. Campbell and Nayeem Islam. " Choices: A Parallel Object-Oriented Operating System". In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[2] Roy H. Campbell and See-Mong Tan. μChoices: An Object-Oriented Multimedia Operating System. In *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, May 1995. IEEE Computer Society.

[3] David Raila and Jishnu Mukerji. A Prototyping Environment for the Choices Operating Systems. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign and Advanced Architecture Department, Unix Systems Laboratories, 1993.

[4] D.R. Engler and M.F. Kaashoek. Exterminate All Operating System Abstractions. In *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 1995. IEEE Computer Society.

[5] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Twelfth International Conference on Distributed Computing Systems*, pages 512–520, Yokohama, Japan, June 1992.

[6] E. Gamma and R. Helm and R. Johnson and J.Vlissides. *Design Patterns, Elements of Object-Oriented Software*. Addison-Wesley, 1995.

[7] J. Mitchell et al. An Overview of the Spring System. In *Proceedings of Compcon 'Spring 1994*, February 1994.

[8] N. Islam. *Customized Message Passing and Scheduling for Parallel and Distributed Applications.* PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., 1978.

[10] See-Mong Tan, David Raila, and Roy H. Campbell. An Object-Oriented *Nano-Kernel* for Operating System Hardware Support. In *Fourth International Workshop on Object-Orientation in Operating Systems*, Lund, Sweden, August 1995. IEEE Computer Society.

[11] K. Thompson. Unix Implementation. *Bell System Technical Journal*, 57(6):1931–1946, July 1978.

[12] W. A. Christopher and S. J. Procter and T. E. Anderson. The Nachos Instructional Operating System. Technical Report UCB//CSD-93-739, University of California, Berkeley, April 1993.